Contract Deliverable for Task 22:

Alternative Technical Architectures for the Integrated Data Base for Mental Health and Substance Abuse Treatment Service Project

Under CSAT/CMHS
Contract # 270-96-007
Project Officer: Jon Gold

June 16, 1997

TABLE OF CONTENTS

LIST OF FIGURES

Executive Summary

This document addresses alternative technical architectures for the Integrated Data Base for Mental Health and Substance Abuse Treatment Service project. We start by defining system architecture, beginning with a well known universe of architectural dimensions and narrowing the dimensions to a set that are appropriate for the current discussion. We describe the problem we are seeking to solve under this contract, and identify and discuss candidate technical architectures in the context of that problem. The logical data model and the application architecture emerge as crucial, interrelated dimensions.

We identify two candidate architectures, open relational database management systems (RDBMS) and SAS, the Statistical Analysis System as likely candidates. The relational database model and its accompanying query language, SQL, are obvious candidates due to their broad acceptance and almost universal availability. The SAS language and it's underlying data model are also identified due to SAS' strong set of procedural tools and data analysis and display capabilities. Other database system architectures are briefly described and rejected.

The relational approach has in its favor ubiquity, an open, interchangeable architecture, an easy to use, well understood data model and a compact storage representation. Major liabilities are potentially high cost, lack of useful data manipulation and transformation tools, and weak analysis and display capabilities.

SAS excels in data manipulation and transformation, has strong analysis and display tools, and has an efficient, if less compact, data storage representation. Liabilities are a proprietary, quasi-relational data model, and a data manipulation language far less known than SQL.

A dual, compromise, architecture is recommended in which SAS is used for intake, transformation, storage and analysis. A highly relational structure, or set of views, is imposed on the final data representation, so that the database or subsets thereof can be exported either in SAS format (transport format) for import by other SAS sites or in relational format (raw data with an ERWin schema), for import by popular RDBMS packages.

System Architectures

The architecture of a system is the plan, embodied in a set of descriptive documents, for producing and maintaining the system over the course of its useful life. Together these plans provide a blueprint of the system to its owners, designers, builders and maintenance staff. For the Integrated Data Base for Mental Health and Substance Abuse Treatment Service project, the system architecture will be developed as a comprehensive plan for the intake, transformation, structuring, representation, analysis and export of the various information sources that will make up the database.

This document addresses alternative technical architectures. The issues are:

- to identify candidate architectures;
- to evaluate them along relevant dimensions; and
- to make a recommendation of a specific technical architecture.

An essential starting point is to determine what aspects of the architecture (plan) we want to focus on in order to identify alternatives. There are a great number of different ways of looking at database systems. We can provide an anchor for the discussion by examining a comprehensive index of system architecture and identifying the elements that are relevant to this stage of the project. The relevant elements will become the dimensions along which we select and identify alternatives.

John Zachman is a well known practitioner in the field of system architecture. The following grid has been interpreted from Zachman's Enterprise Architecture Framework[1]. Each column focuses on a particular aspect of the design, and each row designates a set of design documents that move successively from the general, contextual point of view at the top of the grid to the very detailed, specific point of view at the bottom.

Zachman's framework contains an additional column, not shown, which focuses on motivation[2]. This column was excluded because motivation for the project has already been established by CSAT/CMHS. Although it could be argued that motivation is architecturally relevant, it is clearly not an issue that must be addressed by the project team. Zachman's framework contains an additional row, also not shown in the grid,

[1] Zachman, John A. "Enterprise Architecture: The Issue of the Century" *Database Programming and Design*, 10(3), March 1997.

[2] Zachman, J.A. "A Framework for Information Systems Architecture." *IBM Systems Journal*, 26(3), 1987.

which contains the detailed representations of each column focus (such as source code) which, in our opinion, should be dealt with retrospectively as part of the documentation task.

Each cell in the grid constitutes a separate section in the architectural description envisioned by Zachman. Some of these sections, such as *Goals*, may fit on a single page. Others, such as system design, may be very lengthy. We present Zachman's table here because it goes a long way toward identifying a reasonable universe of potential architectural elements. There are, however, only a few cells that are relevant to the current discussion, and we have identified those cells in **bold** in the table below.

	Data	Function	Network	People	Schedule
Scope (Context)	*Goals* List of things important to the project effort	*Processes* List of processes to be performed	*Locations* List of locations in which operations will be performed	*Organizations* List of organizations involved in effort	*Milestones* List of events significant to implementation
Enterprise Model	*Semantic Model* Diagram of relationship of business entities to data sources	*Business Model* Diagram of business resources used to perform processes	*Logistics Network* Diagram of business locations (nodes) connected by business links (edges)	*Work Flow Model* Chart of movement of work product from organizational unit to unit	*Master Schedule* Timeline of intervals between business events and cycles
Logical System Model	***Logical Data Model*** **Normalized view of data entities and relationships**	***Application Architecture*** **Use case sketch of application components**	***Distributed System Architecture*** **Diagram of hardware nodes & communication links**	*Human Interface* Chart of movement of work product from person (role) to person	*Process Structure* Cycles, updates and movement of data between system components
Physical Technology Model	*Physical Data Model* Databases, tables, keys, indexes, constraints	*System Design* Functional specification of application processes, stored procedures, objects	*System Architecture* Hardware specifications. Vendor supplied software specifications	*Presentation Design* Screen layouts, forms, tabs and menus	*Control Structure* Estimated execution times, system response, record locking strategy

The first row lists descriptions that develop a context for the system. In our case, this context has been set by the RFP, our proposal and the provisions of the contract. The second row lists descriptions that establish the relationships between the business

entities cooperating in the endeavor. Once again, these issues have largely been determined and are laid out in the contract between CSAT/CMHS and The MEDSTAT Group (MEDSTAT) project team, with the notable exception of the cell labeled *Work Flow Model*. In fact, a chart of the movement of work product from organizational unit to unit (especially including producers and consumers of information within the States) would be a very useful document for all parties involved. Its usefulness aside, there seems little need to discuss *alternative* work flow models, as any one of a number of diagrammatic representations would work very well.

Row 3 of Zachman's table contains the essence of the architectural issues we are currently confronting. The logical data model, the application architecture, and the distributed system architecture have not yet been determined, reasonable alternatives exist and we will identify, evaluate and recommend specifics in the sections that follow. Row 4 of the table enumerates documents that provide a level of detail we are not yet ready to address. We will, however, detail the physical data model, system design, system architecture and control structure in our next deliverable, titled *Technical Architecture*, and we'll return to issues of presentation design in future deliverables farther down the road.

Let's look at each of the architectural elements identified above as being relevant to the current discussion. We begin the ***logical data model***. This issue deals with how, logically (or conceptually) the data will be stored. Data will come to us in a variety of formats, but almost always as either ASCII or EBCDIC raw data. Data might either be in flat files (rectangular), or in hierarchical or compressed format, as in variable repeating fields. One option (logical data model) is to leave the data exactly as received. Although this option is easy to implement, it makes the subsequent manipulation and analysis exceedingly difficult, and is therefore not a reasonable alternative. Here are other logical data model options, with notes regarding their situational utility:

1. **ASCII flat (rectangular) files**. A simple representation useful for import into other systems. Easy to edit data. Character data is visible and accessible. Operations on numbers are time consuming because conversion to binary is always required.

2. **ASCII flat (rectangular) files with binary numeric data.** Also simple, but allows faster numeric operations. This is the representation used internally by MEDSTAT's MarketScan data. Requires custom programming for the application architecture, but the investment pays off in speed. Limited options for analysis, suitable for "production line" data manipulation.

3. **Spreadsheet Systems**. Simple, visual, appealing. Systems are inexpensive, and easy to use. Excellent for small amounts of data (few hundred rows) but cannot handle mid-size or large datasets.

4. **Relational Database Management Systems**. They are open, interchangeable, and use a well known data model. Systems are expensive, but produce uniform, easy to manipulate data sources. Optimized for search and update queries.

5. **Non-Relational Database Management Systems**. Examples are Dbase, Rbase, ISAM (Indexed Sequential Access Method) databases, hierarchical and network databases. Generally not open and interchangeable since their data models are proprietary. Queries are much more difficult to formulate. Systems are inexpensive, but typically "top-out" at a particular volume of data (i.e. are PC oriented).

6. **Statistical Analysis Systems**. Examples are SAS, SPSS, BMDP, DATA. Excellent at analysis tasks but typically poor at transformation. Each uses a proprietary internal logical data model, although some incorporate gateways, such as ODBC or relational engines. Systems are expensive, but scale to handle from small to very large amounts of information. Optimized for summarization and analysis.

The preceding list is not exhaustive, but should serve to give the reader a flavor for the types of logical data models that can be employed for data transformation and analysis.

The next relevant element from Zachman's grid is the ***application architecture***. As was hinted at previously, the application architecture is largely determined by the logical data model, at least in most cases. Perhaps the simplest example is to look at data stored in a relational logical data model. It follows that the appropriate application architecture is one centered on SQL, the relational Structured Query Language. If the logical data model is a spreadsheet, such as Lotus 1-2-3, then the application architecture will utilize Lotus macros. If the spreadsheet is Microsoft Excel, the application architecture will utilize VBA, Visual Basic for Applications, Microsoft's application scripting language. If the logical data model is a SAS database, then the application architecture will utilize the SAS data step language. The general principle being articulated here is that, in general, implementations of data manipulation packages bundle a procedural language with the data model, expressly suited to and optimized for the underlying data model.

This principle is not always the case, however. The first two data models described above, ASCII data with and without binary numbers, have no default application

architecture associated with them. It is up to the developer to determine what application architecture is most appropriate given particular design goals.

The last relevant element from Zachman's grid is the ***distributed system architecture***. This design element determines how tasks will be partitioned across and between the collection of physical systems that comprise the enterprise hardware environment. Once again, these decisions generally flow from the choice of logical data model and application architecture.

For example, almost all distributed systems based on the relational data model employ some variant of the client/server design, starting with two-tier designs and moving through multi-tier for larger, more complex systems. This is because the relational data model and its associated SQL language excel at centralized execution of compact standardized queries (but have no user interface capabilities), creating a natural interface point with which to divide systems into a central SQL *server*, connected over a network to multiple *client* programs supplying the user interface.

There are counter-examples to this principle, but for the two data models and application architectures we will examine most closely, distributed system architecture will follow quite naturally.

The Problem to be Solved

In this section, we describe the problem to be solved. Very generally, we wish to combine and integrate Medicaid and non-Medicaid sources of MH/AOD data obtained from three states, Washington, Delaware and Oklahoma in order to facilitate summarization and analysis of costs and utilization in their respective programs. A previous deliverable, the Database Inventory[3], described the variety, content and size of various data sources within the States. A diagram summarizing the data sources, by State, is shown in Figure 1 on the following page.

There are two significant problems that will have to be overcome in order to accomplish this goal. The first is determining a logical structure for the resulting databases that will allow all of the data obtained to be incorporated while remaining simple and regular enough to enable the same or similar data summarization to take place within and across States. The second is matching individual records that occur in Medicaid and non-Medicaid settings. This is difficult because there is no universal identifier that can be used to control the match, and those identifiers that are present and can be used, such as name, address and social security numbers, are unreliable, duplicative and subject to errors.

The difficulty of the first process was raised at the first expert panel meeting in Washington DC in March, 1997. At that time a suggestion was floated that all data be classified into one of three categories: clients, events and services, with one-to-many relationships linking clients and events and linking events and services. The suggestion was very well received by all of the participants at the meeting. Although this representation is not perfect with regard to capturing the detail inherent in all of the data sources inventoried, it works for most and is conceptually simple. At this point in time, we are preceding under the assumption that we will restructure incoming data according to the categories and relationships described above.

The second process involves attempting to match each Medicaid ID record with each Non-Medicaid record and then scoring the tentative match result, that is, determining the likelihood that the two ID records represent the same patient. Logically, we will combine every record from one source with every record from the other source to produce the Cartesian product of the two sources and rank the results. In actual practice, it will not be necessary to produce a complete reflexive join, because we will want a match on at least one of four main identification variables:

[3] "Database Inventories and Profiles of State Mental Health, Substance Abuse, and Medicaid Programs: Delaware, Oklahoma and Washington". Contract Deliverable for Task 21. The MEDSTAT Project Team. May 15, 1997.

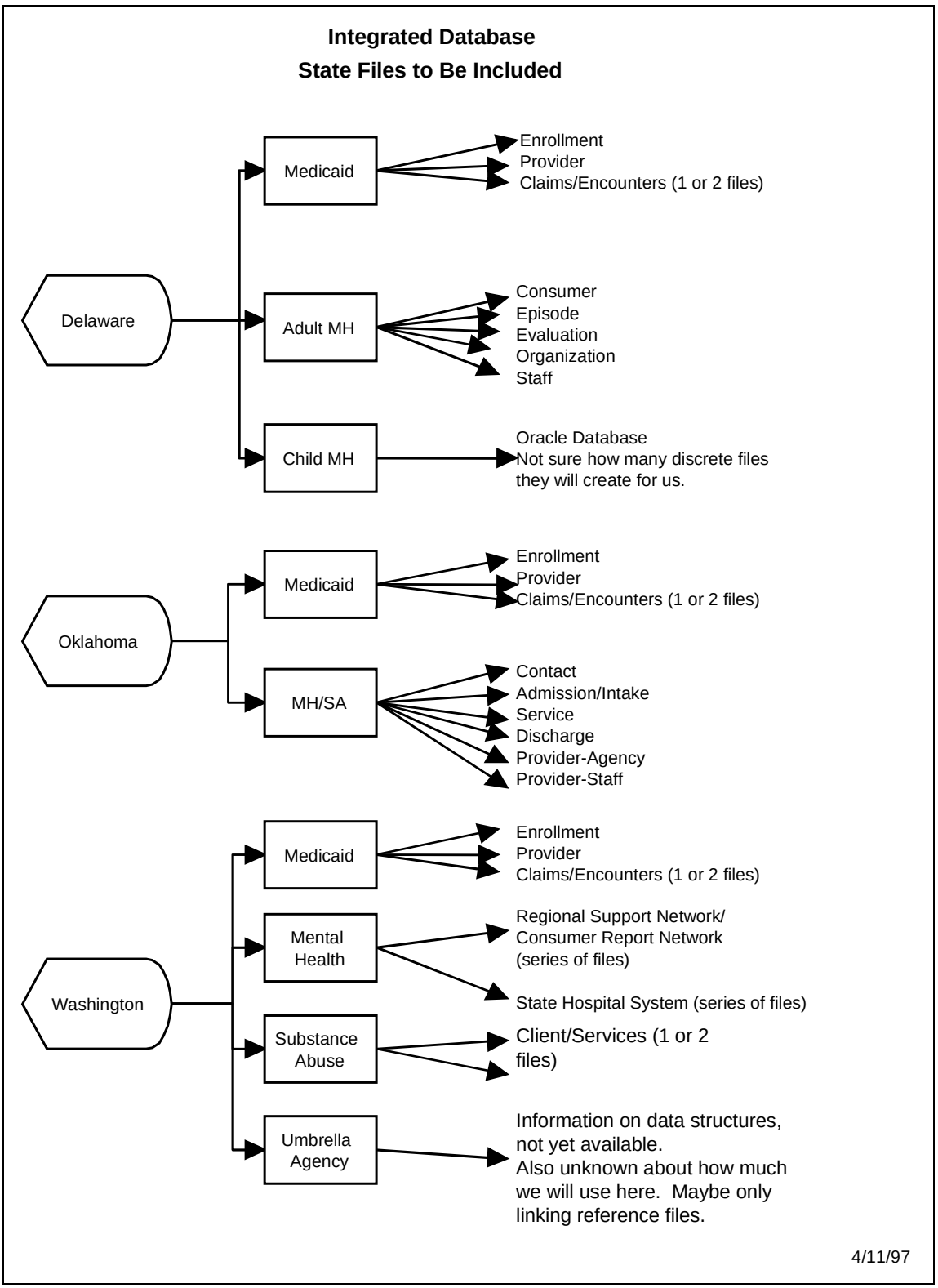

Figure 1. Data Sources by State

1. Medicaid ID (PIC code)
2. Social Security Number
3. Date of Birth and Gender
4. First and Last Name

Matches on additional main variables (including imperfect matches) and matches on other variables, such as race, ethnicity, and ZIP code, will be used to score the match.

The data will be combined at the ID level, without any service data. For the Medicaid data, this will be created from the Medicaid Eligibility file, while the non-Medicaid data will use a demographic extract data set created in the early stages of processing.

Given the assumption that we will transform our incoming data into the clients, events, and services structure described above, and that we will attempt to match individuals across the Medicaid and non-Medicaid domains, we have sketched out a tentative process for the effort.

Figure 2 is a diagram of the tentative plan for matching individuals and processing State data into client, event and service files. Figure 3 is a brief narrative that describes the steps in the diagram. We are not intending to convey that the process is final, complete and comprehensive, because it certainly is not. The process will undoubtedly undergo numerous revisions as we learn more about the data sources and the subsequent analysis plan. We present this information because we want to demonstrate that **the first stage of database construction is not merely loading data**. The task is much more difficult and complex than that, and will require numerous programming and processing steps. This fact, more than any other, will impact our choice of a logical data model and application architecture. We will need to choose an architecture which will allow us to accomplish this process in the simplest, least labor intensive way possible.

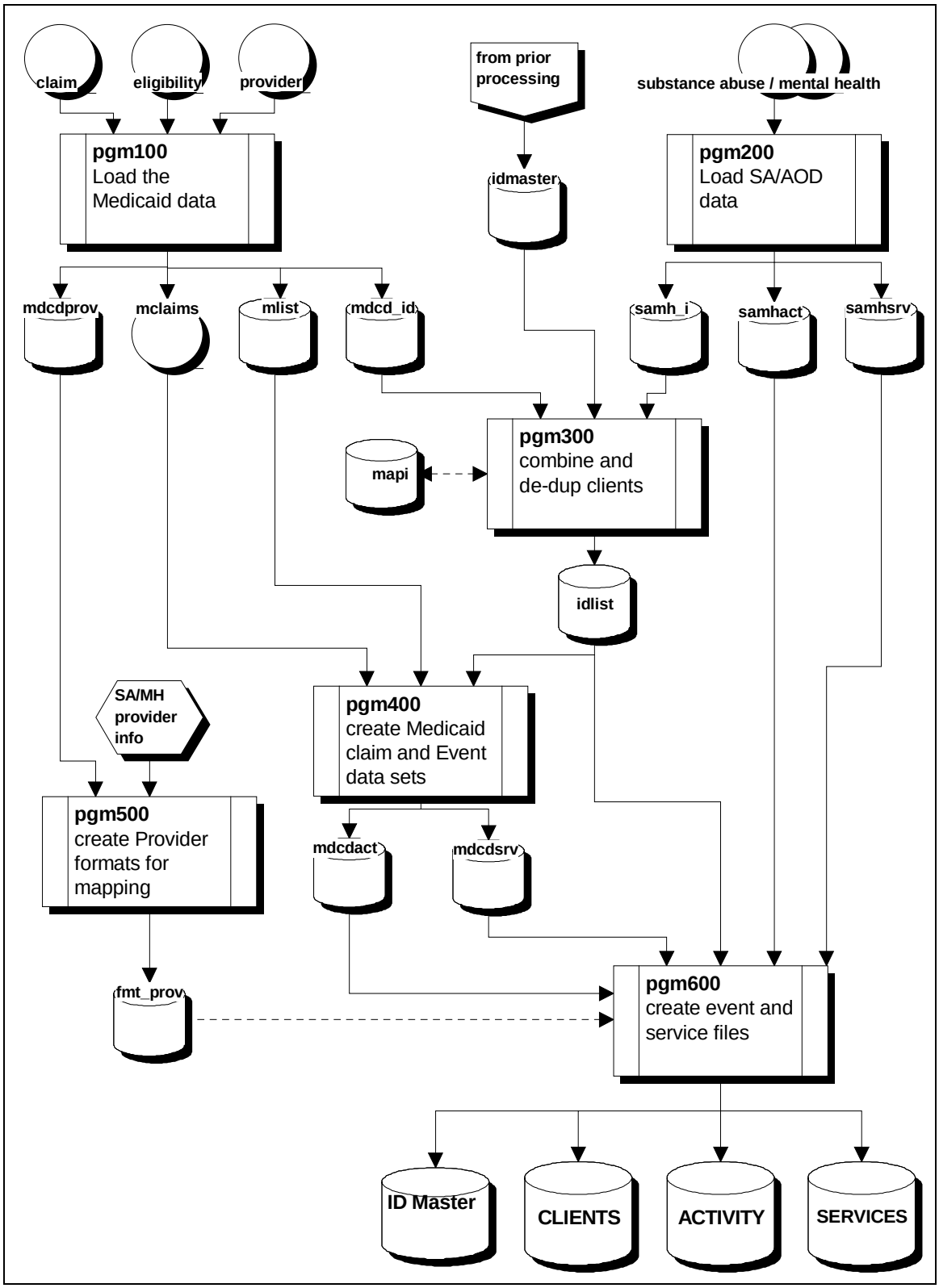

Figure 2. Integrating Medicaid and MH/AOD Data, Diagram

STEP100: Performs a match-merge by Medicaid ID (when available)

- scores the matches
- stores the results for later processing

STEP200: Performs a equi-join where Social Security Numbers (when available) match

- scores the matches
- sorts the data by State ID and score
- keeps the first obs for each State ID
- sorts the data by Medicaid ID
- stores the results for later processing

STEP300: Performs an equi-join where there is a match with birth date, gender or name

STEP400: Scores the joins

- sorts the data by State ID and score
- keeps the first obs for each State ID
- sorts the data by Medicaid ID
- stores the results for later processing.

STEP500: Interleaves merge, join results (steps 10, 20 & 40) by Medicaid ID and score

- sorts by State ID and SCORE
- keeps the first obs for each State ID
- stores the combined data for later processing
- merges the results with the original non-Medicaid ID data to find patients not in Medicaid
- stores the non-Medicaid IDs for later processing.

STEP600: Performs a reflexive join on the non-Medicaid Ids

- keeps matches on name or DOB & Gender
- scores the joins
- sorts by State ID and score
- keeps the first obs for each State ID
- interleaves the results with the combined data from step 50
- match merges the results with the non-Medicaid ID
- stores the results for the next step.

STEP700: Performs a match merge with the CLIENT database from the prior run

Figure 3. Integrating Medicaid and MH/AOD Data, Narrative Description

Identifying Candidate Architectures

In this section, we identify candidate architectures for the logical data model, the application architecture, and the distributed system architecture. As was described previously, the choice of logical data model largely determines the application architecture that is most appropriate to use, and that in turn determines the distributed system architecture.

The Relational Model

We look first at the relational data model. The relational model is an obvious candidate because of its wide acceptance and broad utilization. Ten years ago there was still debate over the data models that lie at the foundation of database management systems, the contenders being network, hierarchical and relational models. No one debates this topic anymore, as the relational model has come to dominate data base technology. The relational model's rigorous mathematical foundation provides a basis for logically proving the correctness and consistency of fundamental database operations such as insertions, deletions and modifications to complex data structures. Other models rely on concepts of pointers, recursive hierarchies and directed graphs; abstractions familiar to computer scientists but opaque to the majority of database users. The relational model's rectangular table structures linked by *relations*, or common data values, are far more intuitive and accessible than the complex structures and linkages inherent in other models. With the exception of experimental and exotic designs, virtually all modern commercial database management systems utilize the relational model.

Another reason the relational model is an obvious candidate is that some of our data contributors, that is the States, employ relational databases to hold some of the databases they will be supplying to us. That makes for convenient import, and assures that exported data will be readable as well.

SAS

A second obvious candidate for our tasks is SAS, the Statistical Analysis System. Born in the seventies as a product that merged statistical processing with a data manipulation language, it has achieved broad acceptance and has largely displaced competing statistical systems like SPSS and BMDP. It is embraced by both statisticians impressed with the rigor of its statistical calculations, and analysts and programmers charged with the typical preliminary tasks of data merging, cleaning and manipulation. Unlike other statistical packages that support only limited data transformation via interpreted commands, SAS contains a full fledged, general purpose programming language. SAS

provides a very workable framework for the always present tasks of linking separate data sources into analytical files. SAS is also used by some of our State collaborators for data manipulation and analysis, once again affording the import/export advantages described above. SAS has been and continues to be widely used at MEDSTAT for tasks very similar to the one we confront here.

Other Candidates

There are a number of other candidates for our task, but none that we will seriously consider. Some of these were mentioned earlier in connection with logical data models. Data can be stored in **raw data format**. A number of very large health databases (including HCFA's Medicaid Tape-to-Tape and SMRF databases) have been stored and manipulated as raw data. The huge disadvantage is the amount of work that must be done writing programs to efficiently manipulate raw data. It is necessary to use a general purpose programming language like C/C++ on a PC or Unix machine or PL/I on a mainframe to efficiently process raw data and this is terribly time consuming. It is estimated that over 100,000 lines of PL/I code were written in the course of the Tape-to-Tape project. Although there are gateways from some data management programs to raw data (for example, Microsoft provides an ODBC driver for raw data) such approaches are grossly inefficient and cannot be considered for our purposes.

PC databases such as **RBASE, Dbase, Access and Paradox** are compelling because of their ease of use and low cost. However, they have a number of faults for our purposes. First, they cannot handle the large volumes of data we will receive. Second, they have poor data manipulation facilities and we would need to rely on C/C++, which has the drawbacks stated above. Third, their reporting capabilities are limited.

Lastly there are a number of so-called Fourth Generation Languages, or 4GL's, that might be used, such as Progress, Clarion Developer and Focus. These languages typically combine a very high level procedural language with an underlying proprietary data model, although some, such as Progress, incorporate native gateways to popular relational databases like Oracle as well. While useful, these packages are typically expensive. They are not widely used, which limits their use to a few expert technicians. Since they are not currently used at CSAT/CMHS, at MEDSTAT or by our State collaborators, we will not explore them further.

Relational Databases - Pros and Cons

The relational database model has a lot to like. It is widely used and understood by almost all database practitioners. There are numerous commercial packages available that all operate on variations of the same basic theme. These packages range from stand-alone PC systems, such as Microsoft Access, through Unix flavors such as Ingres and mainframe implementations like IBM's DB2 all the way to high-end, massively parallel supercomputer implementations like Teradata. Many RDBMS packages, like Informix and industry leader Oracle, run on everything from PC's to mainframes, and even DB2 runs under AIX, IBM's Unix.

Relational database systems are often used for transaction processing and since the operations are performed in real time, they are described by the acronym *OLTP*, for on-line transaction processing. The relational model, when it is applied properly to a database problem, serves to decompose a collection of data into multiple tables that embody *normal* form. There are five levels to the normal form, and a database in fifth normal form has been so thoroughly deconstructed that the typical database operations of insert, delete and update touch as little data in as few tables as possible, which is key to the rapid processing of transactions. Thus the relational model is ideally suited to transaction processing, where numerous individual requests make changes to an exceedingly small fraction of the data, or a query resembles finding a needle in a haystack. This is the relational model's strongest asset.

The relational model and its SQL language are powerful enough to accomplish a variety of data transformation and aggregation tasks, and SQL makes it possible to express these transformations and aggregations in a concise way. Unlike transaction processing, data summarization tasks touch most or all of the data with a single query, examples being consolidation reporting, crosstabulation and univariate or multivariate statistical analysis. The relational tool for putting normalized (decomposed) tables back together again is the *join*. Analytical tasks typically require numerous complex joins to summarize data. The join is computationally expensive, and relational query optimizers typically spend most of their effort optimizing joins, which can take a good deal of time to execute.

This fact has given rise to products targeted at on line *analytical* processing, or *OLAP* for short. This is where the relational model has to stretch a bit to meet the task. There is continuing debate as to whether a single database product can efficiently support both transaction processing and analytic activities. Pure OLAP proponents, pushing their products as competitors to the RDBMS vendors, make a strong case for separating the two, arguing that a design can be optimized for rapid transaction processing or rapid

analytical summarization, but not both. Many relational vendors, Oracle for example, disagree, and have countered with ROLAP, for *Relational* On Line Analytical Processing, and are aggressively pushing their products into the OLAP realm.

As the reader may have assumed, the task we are confronted with much more closely resembles OLAP than it does OLTP, since our processing is dominated by loading, editing, matching, joining and summarizing data. To be fair, Oracle, when augmented with expensive OLAP add-ons does a pretty good job at both. Vendor SYBASE is working on an OLAP add-on module that will restructure relational data in a *transposed* format, that is, by column instead of by row. Theoretically this technique has great promise, but the product is untested and still under development. Other RDBMS systems we are familiar with range from mediocre to poor in OLAP performance. This is a drawback, for our purposes, of the relational data model.

Another drawback is the lack of robust data summarization and analysis measures. Most versions of SQL, the relational query language, support only limited data summarization measures. Neither SQL 89 nor the more recent SQL2 supports univariate statistics other than sum, average (mean), min, max and two versions of count (or N)[4]. This is grossly inadequate for even unsophisticated univariate data summarization, where median and mode, percentiles, and measures of dispersion such as standard deviation, skewness and kurtosis are commonly employed.

SQL does allow for one-way and n-way frequency distributions, but not for statistical measures associated with n-way tables, such as chi-square or t tests.[5] Clearly, SQL was not designed for statisticians, and it is even a stretch to say that it supports data analysis, at least much beyond counts and sums.

In addition, although SQL can be used to create n-way frequency distributions, it cannot display them in any but the most rudimentary of formats. The PIVOT statement, used in Microsoft Access SQL to create crosstab tables, is a non-standard extension to SQL that is not supported in other SQL dialects.[6] Many, if not most, relational database management systems either incorporate or have optionally available *report writing* modules to allow for flexible display of data. Of course, these report writers and their related languages are different between RDBMS implementations, negating the advantage of SQL's universal syntax. Lack of standard facilities for data summarization and display is a major drawback of the relational model for our purposes.

[4] Groff and Weinberg. *LAN Times Guide to SQL*. McGraw Hill, 1994.
[5] Frank Lusardi. *Database Experts Guide to SQL*. McGraw Hill, 1988.
[6] *Microsoft Access Language Reference*. Microsoft Press, 1995.

Another problem with the relational model and SQL is the lack of a procedural language for manipulating the underlying data. SQL is very good at what it does, but when you reach the limits of it's capabilities, you have no alternatives. Typically, relational database vendors have finessed this problem by implementing *cursors*, a movable pointer used to access data in a particular table row in the API (applications programming interface) for their product. Thus programmers can bypass SQL and move row by row through a table or query result set reading and writing data at will. This solves the problem, but in so doing we are back to the labor intensive job of coding in C/C++, PL/I or some other programming language.

Another problem with the relational application architecture as it is delivered by vendors is the lack of tools for data input and transformation. RDBMS packages typically contain a tool for bulk data loading which is much faster and more flexible than SQL. While these tools may allow for simple transformations and reformatting, they do not contain facilities for things like unpacking variable length records, reading binary and other mainframe style data formats, translating fields from EBCDIC to ASCII, transforming dates and times from one format to another and other typical tasks.

In summary, while relational databases provide a compelling data model for a variety of reasons, their associated application architectures have serious drawbacks for data summarization and display, as well as data manipulation and transformation.

SAS - Pros and Cons

The SAS data model and application architecture have a number of advantages for our purposes. Like the best of the relational databases, it is capable of handling large quantities of data, and can do so very efficiently. Originally written in IBM assembler which accounted for its outstanding performance, in the 80's it went through a substantial architectural renovation and was completely rewritten in C, a prescient move that set the stage for an ambitious cross platform porting spree, with the result that SAS probably runs on more diverse hardware and operating system platforms (although Oracle runs a close second) than any other database systems. SAS runs on virtually all PC, Unix and mainframe computers.

SAS excels at import and export of raw data. The SAS data step incorporates a high level language that is somewhat of a cross between PL/I and C and a simple, rectangular underlying data model. Unlike SQL, the language is procedural, allows arbitrary bi-directional movement through data, and implements *state*, or memory (such as flags, counters and accumulators) between rows. This is a much cleaner and easier alternative to using cursors in a high level language via the relational database API.

SAS provides a very workable framework for the always present tasks of linking separate data sources into analytical files. This framework consists of sorting and *merging* separate files by means of common linking variables, a technique congruent to the relational join. Variants of this basic merge operation are allowed to the extent that there are direct SAS analogs of relational equi-joins, left and right inner joins, and outer joins, or Cartesian product.

So similar is the underlying SAS data model to the relational model that SAS was able to implement a PROC SQL, in which standard SQL syntax can be applied to native SAS files. Additionally, SAS has implemented a collection of relational database access engines tied to vendor specific RDBMSs such as Oracle, Informix and Sybase. These engines allow SAS data steps and procedures to read and write native relational tables.

SAS incorporates a huge library of transformation and conversion functions that cover almost all data formats and structures. Reading and writing variable length records and repeating fields in SAS is easier than in COBOL, where almost all such transgressions against the laws of clean data structure usually originate.

One of SAS' strongest points is that it contains a variety of excellent procedures for the display of data. SAS execution is divided between data steps, which are user written programs that read, write and transform data, and PROCs, which are pre-written

procedures used to summarize, analyze and display data. A few of these procedures are described below.

- **PROC FREQ**: Used to create one-way to n-way frequency distributions in a variety of formats. Includes appropriate statistical measures for one-way and two-way tables.

- **PROC TABULATE**: Used to create tabular displays. Allows for the arbitrary partitioning of a data source by categorical variables, and the application of a full set of univariate statistics (including median and mode) to the subsets within partitions. Partitions can be laid out within column, row or page dimensions.

- **PROC UNIVARIATE**: Applies the full range of univariate statistics to a data set, including percentiles and measures of dispersion.

- **PROC MEANS**: similar to UNIVARIATE, but with an abridged set of statistics. Faster than UNIVARIATE and with the option to write means and other measures back to the data set, it is useful for tasks like creating Z scores or deviations from the mean.

- **PROC PLOT, PROC CHART and SAS/GRAPH**: Plot and Chart perform their respective tasks with arbitrary display devices. Graph makes use of high resolution displays and printers for detailed graphics.

Perhaps the biggest disadvantage of SAS from our point of view is that it is less widely used and accepted than the relational data model, having a much smaller installed base than the leading relational software product, Oracle. This limits opportunities for native import and export of data, although between MEDSTAT and CSAT/CMHS and at least some of the agencies we'll work with in the States, those opportunities do exist.

Another drawback is that the SAS logical data model is a less compact and more redundant representation of data than the very clean normal form representation used in relational databases. It should be noted that SAS databases can be decomposed into normal form, just as relational databases can be and often are *denormalized* for OLAP tasks. We'll return to and make use of this fact in the Conclusion. The issue is that the SAS and relational application architectures are optimized for one or the other data representations. For example, SAS uses the mechanism of arrays, collections of similar variables that are iteratively accessed, which would be taboo in a normalized relational database. Although the SAS data representation is less elegant than a normalized relational one, it is practical and useful for data summarization and analysis.

In summary, we feel that the SAS logical data model is adequate for our purposes. The data manipulation and transformation capabilities are unsurpassed and crucial for this project, as are SAS' data display and analysis functions.

Conclusion

We have examined two specific architectures in depth. The relational approach has in its favor ubiquity, an open, interchangeable architecture, an easy to use, well understood data model and a compact storage representation. Major liabilities are potentially high cost, lack of useful data manipulation and transformation tools and weak analysis and display capabilities. SAS excels in data manipulation and transformation, has strong analysis and display tools, and has an efficient, if less compact, data storage representation. Liabilities are a proprietary, quasi-relational data model, and a data manipulation language far less well known than SQL.

As the reader may have guessed by this point in the discussion, we are leaning strongly to the use of SAS for intake, transformation, storage and analysis. Our only misgiving is that the SAS logical data model is not as clean and parsimonious as its relational counterpart and that there is not nearly as much use and knowledge of SAS as there is of various relational databases, hindering database export activities.

With this in mind, we would like to recommend a dual, compromise architecture. We propose that SAS be used for intake, transformation, storage and analysis, but that we produce a normalized relational format for data export. This makes best use of SAS' strengths, while keeping options for data dissemination open.

A relational structure, implemented as a set of *views* and data step transformations, will be imposed on the final data representation, to bring it into third normal form. In practice, this will be accomplished by taking the typically wide SAS rows that focus on multiple entities and contain repeating fields and creating narrower tables, with unique primary keys and relational links to other tables. We propose to use ERWin, an entity relationship structuring tool, to create the relational schema and to provide a mechanism for documentation and easy import of the resulting raw data tables into a variety of relational database packages.

In this way, the database or subsets thereof can be exported either in SAS format (transport format) for import by other SAS sites or in relational format (raw data with an ERWin schema), for import by popular RDBMS packages.

We look forward to CSAT/CMHS's comments and feedback and would be happy to discuss or elaborate on any of the issues presented here.